
ABSTRACT

Distributed deep learning frameworks like Federated learning (FL) and its variants are enabling personalized experiences across a wide range of web clients and mobile/IoT devices. However, these FL-based frameworks are constrained by the computational resources at clients, due to exploding growth of model parameters (eg. billion parameter model). Split learning (SL), a recent framework, reduces client compute load by splitting the model training between client and server. This flexibility is useful for low-compute setups but is achieved at cost of massive increase in bandwidth consumption and often results in sub-optimal convergence, especially when client data is heterogeneous. In this work, we introduce AdaSplit which enables efficiently scaling SL to low resource scenarios by reducing bandwidth consumption and improving performance across heterogeneous clients. To capture and benchmark this multi-dimensional nature of distributed deep learning, we also introduce C3-Score, a metric to evaluate performance under resource budgets. We validate the effectiveness of AdaSplit under limited resources through extensive experimental comparison with strong federated and split learning baselines. We also present a sensitivity analysis of key design choices in AdaSplit which validates the ability of AdaSplit to provide adaptive trade-offs across variable resource budgets.

ACM Reference Format:


1 INTRODUCTION

Distributed machine (deep) learning is characterized by a setting where many clients (web browsers, mobile/IoT devices) collaboratively train a model under the orchestration of a central server (eg. service provider), while keeping the training data decentralized.

As strict regulations emerge for data capture and storage such as GDPR [23], CCPA [77], distributed deep learning is being used to enable privacy-aware personalization across a wide range of web clients and smart edge devices with varying resource constraints. For instance, distributed deep learning is replacing third-party cookies in the chrome browser for ad-personalization [10, 20], enabling next-word prediction on mobile devices [29], speaker verification on smart home assistants [26], HIPPA-compliant diagnosis on clinical devices [68] and real-time navigation in vehicles [19].

Figure 1: AdaSplit achieves improved accuracy under limited resources (bandwidth & compute) and can adapt to variable resource budgets. Results on Mixed-NonIID dataset.

A general distributed deep learning pipeline involves multiple rounds of training and synchronization steps where, in each round, a model is trained with local client data and updates across multiple clients are synchronized by the server into a global model. Techniques have been proposed with the goal to maximize accuracy under constraints on resource (bandwidth, compute) consumption. Figure 1 compares our proposed AdaSplit (in yellow) with strong baselines [27, 35, 49, 56, 79, 86] along these dimensions.

Federated learning (FL) [56] is one of the widely studied frameworks [12, 31, 49, 56, 86]. In each round of FL, first, all clients train a copy of the model locally on their device for several iterations and then, communicate the final model parameters with the server which synchronizes updates across clients by averaging all clients model parameters and shares back the unified global model for next training round. This is summarized in Figure 2. With entire model training on-client, federated learning is challenged by the compute budgets of client devices. First, on-client model training needs resource-intensive clients (with high-performance GPUs to avoid stragglers) and is increasingly becoming impractical due to exploding growth in model sizes (eg. billion parameter models for language and image modeling [17, 66, 89]). Second, as the number of clients (and/or model sizes) scales, bandwidth requirements for the system may worsen as entire models need to be communicated between client and server. Furthermore, at inference, often, storing the entire trained model on-client can have intellectual property implications that limit real-world usability.

More recently, the split learning (SL) framework [27, 65, 79, 82, 83] has emerged to alleviate some of these concerns in federated learning. SL reduces client computation load by involving the server in the training process. In each round, clients take turns to interact with the server for multiple iterations where they update parameters of a local model on the client and a (shared) global model residing on the server. Specifically, at each iteration, the client model generates input activations that are communicated to the server which uses them to compute gradients that are used to train the server model and transmitted to the client to train the client model. This is summarized in Figure 2. While client computation is significantly reduced in SL than FL, this comes at cost of an increase in client-server communication and often in-efficient convergence. First, since the client is dependent upon the server for training gradients, required communication budgets increase as the client interacts...
with the server in every iteration of a round (vs once-per-round in FL). This server is also blocked to train synchronously with each client. Second, since clients sequentially update shared parameters on the server, convergence may be in-efficient or sub-optimal, especially when client data is heterogeneous. Alleviating these concerns is the focus of this work.

We introduce AdaSplit, which enables split learning to scale to low-resource scenarios. First, a key insight in AdaSplit is to eliminate client dependence on server gradient which reduces communication cost and also enables asynchronous (and reduced) computation. Next, motivated by the fact that neural networks are vastly overparameterized, AdaSplit is able to improve performance by constraining heterogeneous clients to update sparse partitions of the server model. As shown in Figure 1, this enables AdaSplit to not only achieve improved performance under fixed resources (higher accuracy when similar bandwidth and compute) but also adapt to variable resource budgets (the trade-off curve). Furthermore, to capture and benchmark this multi-dimensional nature of distributed deep learning, we also introduce C3-Score, a metric to evaluate performance under resource budgets.

The contributions of this work can be summarized as:

- We introduce AdaSplit, an architecture for distributed deep learning that can adapt to and improve performance across variable resource constraints.
- We introduce C3-Score, a metric to benchmark and compare distributed deep learning techniques.
- We validate the effectiveness of AdaSplit through experiment comparison with state-of-the-art methods and sensitivity analysis of different design choices.

2 PRELIMINARIES AND MOTIVATION

First, we formalize the protocol and notation for the split learning (SL) framework. This is also visualized in Figure 2. For completeness, we also visualize the FL protocol in the same Figure. While the training protocol may appear different, we unify their design choices along key dimensions. Next, we formulate three key design dimensions and contextualize specific choices of FL and SL. This helps motivate our proposed AdaSplit technique.

2.1 Split Learning

Consider a distributed learning setup with N participating clients and 1 coordinating server. The key idea of split learning (SL) is to distribute (or split) the parameters of the training model across client and server. Each client i, for i ∈ [1, 2, ..., N] is characterized by a local client dataset Di, local client model Mi and a single server model M0 which is updated by all the clients. The training protocol is executed over R rounds of T iterations each. In each round, the N clients sequentially obtain access to interact with the server for model training over T iterations. In each iteration j (for j ∈ [1, 2, ..., T]) of client i updates the parameters of Mi and Mi. First, a mini-batch (xi, yi) is sampled from Di and passed through layers of client model Mi to generate activations ai (= Mi(xi)). In this document, we may refer to ai as split activations. Second, the pair of (ai, yi) is transmitted to the server. Third, at the server, ai is passed through layers of server model M0 to generate predictions yi (= M0(ai)). The loss function L(yi, yi) is computed to generate gradients which are used to locally update parameters of M0 and then transmitted to the client to update parameters of Mi. In the classical setup, clients follow a round-robin mechanism where client i + 1 can start interacting with the server only after client i has completed its T iterations for the round. The global model is synchronized implicitly across clients by updating weights of the shared server model M0. Furthermore, in some variants, clients models is transmitted between pairs of clients during a round [27] or averaged over all clients after the round ends [79]. Extensive work has been conducted to establish privacy in split learning and, while beyond scope of this paper, we briefly discuss that in Section 7.

2.2 Design Dimensions: 3C’s

We define three key design dimensions which focus on how i) model is trained on local client data (Computation) and, ii) updates across the clients are synchronized, via the server, into a global model (Communication and Collaboration).

1. Computation: This governs how the processing of data at each client is executed between client and server. Hence, the computation cost can be defined as the total sum of floating-point operations (FLOPs) executed on the client and server. For N clients, this cost (C1) can be represented as:

\[ C1 = \sum_{i=1}^{N} R \ast (F^c_i \ast T^c_i + F^s_i \ast T^s_i) \] (1)

where, \( F^c_i \) are the FLOPs executed on client for \( T^c_i \) iterations, \( F^s_i \) are FLOPs executed on server for \( T^s_i \) iterations when training with data for client i and R is number of rounds. \( F^c_i \) and \( F^s_i \) increase (or decrease) monotonically with increase (or decrease) in size of client model \( M_i \) and server model \( M^0 \) respectively. In federated learning, \( F^c_i = 0 \) and \( T^c_i = 0 \) since the entire model is executed on client device (\( M^0 = 0 \)). In contrast, split learning allows to split the model and distribute \( F_c \) and \( F_s \) between client and server, based on resource availability. This flexibility of split learning allows scaling to low-resource setups where clients are compute constrained (but servers may scale horizontally) and is a key aspect for design of AdaSplit. However, we also note that this classical split learning framework increases compute load on the server and also blocks the server to train synchronously with each client.

2. Communication: This governs how client-and-server interact with each other. Hence, the communication cost can be defined as the total payload that is transmitted between each of the N client-server pairs over multiple rounds of training. Federated and Split Learning differ based on the modality of the payload and frequency of interaction. However, without loss of generality, this cost (C2) can be represented as:

\[ C2 = \sum_{i=1}^{N} \sum_{j=1}^{R} \sum_{k=1}^{T} (P_{is} + P_{si}) \ast \sigma(i, j, k) \] (2)

where \( N \) is number of clients, \( R \) is training rounds and \( T \) is iterations per round. \( P_{is} \) is the payload transmitted from a given client i to server s and \( P_{si} \) is the payload transmitted from server s to client i. \( \sigma(i, j, k) \) denotes if client i interacts with server during iteration k of round j. In federated learning, client-server interact using model weights once-per-round. Hence, size of each \( P_{is}, P_{si} \) is size of the total model and \( \sigma(i, j, k) = 1 \) only for \( k = T \) (last iteration of every
In federated training, the global model in a round \( r \) and consequently updated client models \( M^r_i \) are obtained as:

\[
M^r = \sum_{i=1}^{N} (M^r_i \ast p^r_i)
\]

\[
M^r_i = M^r \forall i \in [1, 2, ..., N]
\]

where \( p^r_i \) is a weight assigned to client \( i \) in round \( r \).

In split training during each round \( r \), the server model \( M^r_s \) is updated sequentially by all client \( i \) for all \( i \in [1, 2, ..., N] \) as:

\[
M^r_s = M^r_s - \alpha \ast \nabla L(M^r_i(a_i), y_i)
\]

In some variants of split learning such as [79], local client models may also be synchronized, at the end of each round, similar to federated learning using equation 3. Then, the global model is obtained by stacking the server and client models. We note that when data across clients are non-iid (that is common on real-world distributed setup), inefficient or sub-optimal converged accuracy is observed in \( M^s \). In split learning, we posit that this happens since gradients from non-iid activations sequentially update the same parameters which violates assumptions of empirical risk minimization [81].

## 3 ADASPLIT

In this section, we delineate the design choices of AdaSplit along each of the three dimensions. We also discuss corresponding tradeoffs that enable AdaSplit to adapt to variable resource constraints. Unless specified otherwise, we follow the same notation as defined in Section 2. The architecture is visualized in Figure 2.

### 3.1 Computation

The training model is split between the client and server. Following split learning, each client is characterized by a local client model \( M^c_i \) and a global server model \( M^s \) that is shared across all clients. This flexibility to distribute the model allows scaling split learning (and AdaSplit) to low resource setups. Recall from Section 2.2, that in classical split learning, this increases computation load on the
server and also blocks the server to train synchronously with each client model which depends upon the server for the gradient.

AdaSplit alleviates these concerns by i) eliminating the dependence of the client model on server gradient and ii) only training the server intermittently. This further lowers the total computation cost by decreasing \( T_s \) (compute iterations on the server) and also unblocks the server to execute asynchronously from the client.

**Local Client Gradient:** First, AdaSplit generates the gradient for training client model on-client itself using a local objective function \( L_{\text{client}} \) which is a supervised version of NT-Xent Loss \([76]\). Given an input batch, \( b \sim D_i \), then for each input \((x_i, y_i) \sim b, L_{\text{client}} \) is applied on a projection \((H(.))\) of the activations \( a_i \) generated by the client model \((= M^i_c(x_i))\). Let \( q_i = H(a_i) \) be the corresponding embedding of an input \( x_i \), and \( Q^i \) be the set of embeddings of other inputs with the same class as \( x_i \) in the batch \( b \), the loss can be represented as below:

\[
L_{\text{client}} = \frac{1}{|b|} \sum_{i=0}^{|b|} \log \frac{\exp(q_i \cdot q_s / r)}{\sum_{j \neq i} \exp(q_i \cdot q_j / r)} \tag{5}
\]

Here, \( r \) is a hyperparameter, which controls the "margin" of closeness between embeddings. We set \( r = 0.07 \) in all our experiments.

The pairs (anchor \( q_i \), positive inputs \( q_s \)) required in \( L_{\text{client}} \) are sampled using the ground truth labels \( (y_i) \) locally on client.

**Intermittent Server Training:** Second, AdaSplit also splits the \( R \) round training into two phases: A) **Local Phase** B) **Global Phase**. Local Phase lasts for the first \( k \) rounds when only the client model trains, asynchronously and without interacting with the server, using \( L_{\text{client}} \). After \( k \) rounds (till end), the Global Phase starts where client interacts with the server by transmitting activations. The server model only now starts training on data from the clients. The server model \( M^s \) is optimized using a server loss function \( L_{\text{server}} \) which is cross-entropy \( L_{\text{ce}} \) for classification tasks. We note that, even in global phase, client model continues to train using \( L_{\text{client}} \) and does not receive any gradient from the server.

Essentially in AdaSplit, client models leverage compute resources of the server only when required. AdaSplit can adapt to variable computation budgets by regulating two key hyperparameters: i) size of the client model \( (\mu) \) (for client compute); ii) duration of local phase \( (\kappa) \) (for server compute). We study the specific impact of these design choices in Section 6. Also, in practice, we observe considerable reductions in total computation since \( \kappa \) can assume large values \((0.8R)\), where \( R \) is total training rounds, without significant loss of performance. We corroborate this with results in Section 5.

### 3.2 Communication

Recall from Section 2.2 that in classical split learning, the high client–server interaction can be prohibitive for communication cost. AdaSplit alleviates this problem by i) decreasing the payload size and ii) the frequency of communication.

**Smaller Payload:** First, we note that eliminating client dependence on server gradient also significantly reduces communication cost, apart from decreasing computation. In AdaSplit, \( P_s = 0 \) (from equation 2 in Section 2.2) throughout training for each client \( i \). Through sensitivity analysis in Section 6, we validate that this design choice marginally drops the performance while significantly reducing communication.

**Infrequent Transmission:** Second, we note that two-phase training is also beneficial for communication. In the **Local Phase**, there is no client-to-server communication, with the payload \( P_{si} = 0 \) for all clients \( i \) (from equation 2 in Section 2.2). In the **Global Phase**, clients may start transmitting activations to the server. In this phase, only a subset of clients communicates with the server in each round. Specifically, we introduce an Orchestrator \( (O) \) which resides on the server and uses a running statistic of local client losses to select \((\eta N)\) clients in each iteration, that communicate with the server. In AdaSplit, \( O \) uses a UCB [2] strategy to prioritize clients who need the server model to improve performance on their data (exploitation) while also ensuring that the final model can generalize well to different client data distributions (exploration).

Let \( S^i \) be a binary flag denoting if client \( i \) is selected at iteration \( t \) and \( L^i_t \) denote the server loss from activations \((a_i)\) for the iteration. At each iteration \( t \), selected clients \((i.e., S^i_t = 1)\) transmit input activations to update server model and the loss \( L^i_t \) is stored. For unselected clients \((i.e., S^i_t = 0)\), \( L^i_t \) is defined the average of their loss value in previous iterations \((L^i_t = \frac{L^i_{t-1} + L^i_{t-2}}{2})\). Here, we note that \( L^i_t \) is only used for selection and the client model continues to train locally with \( L_{\text{client}} \). Finally, \( O \) assigns a new score to each client using the advantage function and clients with the top-\( \eta \) scores are selected for next iteration. The advantage function \( (A_i) \) for [2] is defined below:

\[
A_i = \frac{\sum_{t=0}^{T} y^T - 1 - t \cdot L^i_t}{s_i} + \sqrt{\frac{2 \log T}{s_i}} \tag{6}
\]

where, \( li = \sum_{t=0}^{T} y^T - 1 - t \cdot L^i_t \), \( si = \sum_{t=0}^{T} y^T - 1 - t \cdot S^i_t \) and \( T \) is total iterations in the round. \( y \in [0, 1] \) is a hyperparameter that determines the importance of historical losses. We initialize \( L^i_0 = 100 \) for all clients for \( t = 0 \) and \( t = 1 \).

Before proceeding, we make a few statements here. First, we note that subset selection has previously been used in FL to regulate communication cost \([14, 49, 56]\) where the global model after a round may be obtained from few clients only (see variable \( p^f \) in equation 3). However, classical split learning does not have a similar infrastructure since each client is entirely dependent on the server for gradient during training. Eliminating client dependence on server gradient in AdaSplit helps realize the benefit. Finally, we highlight that the design of orchestrator is specialized for AdaSplit where it needs to be invoked in each iteration \((vs \text{ rounds in FL})\) and selects client for training \((\text{vs model averaging in FL})\).

AdaSplit can adapt to variable communication budgets by regulating two key hyperparameters: i) the number of selected clients \( (\eta) \); ii) the duration of the local phase \( (\kappa) \). We study the specific impact of these design choices in Section 6. In practice, we observe considerable reductions in communication cost since \( \kappa, \eta \) can assume large values \((0.8 + R, \eta = 0.6 + N)\) without significant loss of performance. We corroborate this with results in Section 5.

### 3.3 Collaboration

AdaSplit, like split learning, synchronizes updates in the global model by requiring clients to sequentially update shared server model parameters. Recall from Section 2.2 that when inter-client...
data is heterogenous, this often results in the global model converging to sub-optimal accuracy. To alleviate this, the key idea in AdaSplit is to have each client update only a partition of the server model ($M_i$) parameters. The motivating insight is to take advantage of the fact that neural network models are vastly over-parameterized [59] and only a small proportion of the parameters can learn each (client’s) task with little loss in performance [24, 43, 52, 78].

Update Sparse Partitions of Server Model: During the global phase, we add an $L^1$ weight regulator to promote sparsity in the server model $M^s$. Specifically, instead of pruning the network, we learn a client ($i$) specific multiplicative mask $m_i$ which constrains the set of $M^s$ parameters client $i$ can update. Given batch of activations $a_i$ from client $i$, server model $M^s$ is updated as:

$$ M^s = M^s - \alpha \cdot m_i \cdot \triangledown L(M^s(a_i), y_i) $$

This simulates relative sparsity (for each client) in $M^s$ without pruning any parameters since goal is to increase server model capacity (to accommodate many diverse clients) rather than achieving compression. Here, $m_i$ evolves during training and is forced to be extremely sparse by optimizing the following objective on the server for each client $i$:

$$ L_{server} = L_{ce}(y_i, y_i) + \lambda \cdot \omega (m_i) $$

where, $\omega(.)$ is an $L^2$ regularizer and $\hat{y}_i = M^s(M^s(x_i))$. The $\lambda$ hyper-parameter promotes sparsity of the masks and can be intuitively visualized as controlling the extent of collaboration between clients on the server. At inference, the effective server model for client is $M^s + m_i$ where $m_i$ is a highly sparse binary mask and may be stored on client. Results in Section 5 show that this strategy of regulating collaboration significantly improves performance. Finally, we note similarities between each round of collaboration in AdaSplit and continual learning, albeit AdaSplit works in activation space and is iterative. However, we anticipate exploring this connection may present interesting directions of future work.

4 EXPERIMENTAL SETUP

First, we formalize the experimental protocol with datasets and baselines. Next, we define the evaluation protocol and introduce the C3-Score as a unified metric to benchmark and compare the efficiency of distributed deep learning techniques. Finally, we summarize implementation details for results presented in this work.

4.1 Datasets

To robustly validate the efficacy of AdaSplit, we conduct extensive experiments on benchmark datasets and simulate varying levels of inter-client heterogeneity. Specifically, we design two experimental protocols, as described next: a) Mixed-CIFAR: We divide the 10 classes of CIFAR-10 [40] into 5 subsets of 2 distinct classes each. Every client is assigned data from one of the 5 subsets. In this protocol, there is low and consistent heterogeneity between data across all pairs of clients. b) Mixed-NonIID: We use 5 benchmark datasets: i) MNIST [42] ii) CIFAR-10 [40] iii) FMNIST [88] iv) CIFAR-100 [40] v) Not-MNIST [9] and each client receives samples from exactly one dataset. In this protocol, there is high and variable inter-client heterogeneity between client pairs. For instance, clients with FMNIST and MNIST have lower pairwise heterogeneity between each other and high pairwise heterogeneity with clients containing CIFAR-100.

For experiments with both protocols, input images are scaled to 32x32x3 and grayscale MNIST/FMNIST images are transformed by stacking along channels. We will release the evaluation protocol with the code for use by the research community.

4.2 Baselines

We compare state-of-the-art split learning and federated learning techniques. Specifically, for split learning, we compare with SL-bas [27] and SplitFed [79]. To ensure validity of analysis and also highlight efficacy of results, we also compare with state-of-the-art federated learning techniques: FedAvg [56], FedNova [86], Scaffold [35] and FedProx [49]. These techniques are specially designed for heterogenous (non-iid) setups and provide strong benchmarking for the efficacy of AdaSplit.

4.3 Evaluation Metrics: C3-Score

We evaluate performance both independently along each of the dimensions as well as using a unified metric. To evaluate along design dimensions, we report Accuracy, Bandwidth and Compute. Accuracy is measured as mean and standard deviation over multiple independent runs with different seeds. Bandwidth is measured in GB and Compute is measured in TFLOPS. We note that, in real-world cases, servers may scale horizontally and bottleneck is often at client. For completeness, we report both client compute and total (client+server) compute.

C3-Score: For an efficient method, the goal is to maximize accuracy while minimizing resource (bandwidth, compute) consumption. We introduce C3-score, a metric to evaluate joint performance of any distributed deep learning technique in this multi-dimensional setup. Let $B_{max}, C_{max}$ be the maximum resource budgets for bandwidth and client compute. We note that $A_{max}$ = 100% (max achievable accuracy) for predictive tasks. Consider, a given method $m$, with mean accuracy $A_m$, bandwidth consumption $B_m$ and client compute consumption $C_m$. Then, the C3-Score is defined as below:

$$ C3 - Score(A_m, B_m, C_m) = (\hat{A}_m) \cdot e^{(-\hat{B}_m + \hat{C}_m)/T} $$

where, $\hat{B}_m = B_m/B_{max}$, $\hat{C}_m = C_m/C_{max}$, $\hat{A}_m = A_m/A_{max}$ and $T$ is a scaling temperature.

The C3-Score always lies between 0 and 1 with a higher score representing a better (more efficient) method. Trivially, C3-Score $\rightarrow 0$ as $\hat{B}_m \rightarrow \infty$ or $\hat{C}_m \rightarrow \infty$ (as consumption increases or budget decreases). Here, we make a few comments about the metric. First, considering resource budgets is useful for real-world use and can also enable to directly eliminate methods that exceed the budget. We take motivation from work in differential privacy (privacy budgets) which contextualize comparing between methods. Also, normalizing cost with the budget allows realizing a bounded metric which is useful for comparison and benchmarking. Finally, we assume a multiplicative form to the metric to enable an easy extension to other resource dimensions in the future. Specifically, incorporating privacy budgets is an interesting direction of future work.

4.4 Implementation Details

All experiments are trained for (R=20) rounds with 1 epoch per round using the same convolutional (LeNet) backbone. Results are reported for 5 (=N) clients. For the federated learning baselines,
Table 1: Results on Mixed-NonIID dataset. AdaSplit achieves improved performance while reducing resource (bandwidth, compute) consumption. This is corroborated by the C3-Score (higher is better).

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>Bandwidth (GB)</th>
<th>Compute (TFLOPS)</th>
<th>C3-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL-basic</td>
<td>84.65 ± 0.32</td>
<td>84.54</td>
<td>3.76 (15.14)</td>
<td>0.72</td>
</tr>
<tr>
<td>SplitFed</td>
<td>84.67 ± 0.24</td>
<td>84.64</td>
<td>3.76 (15.14)</td>
<td>0.73</td>
</tr>
<tr>
<td>FedAvg [56]</td>
<td>82.21 ± 0.19</td>
<td>2.39</td>
<td>17.13 (17.13)</td>
<td>0.72</td>
</tr>
<tr>
<td>FedProx [49]</td>
<td>85.09 ± 0.29</td>
<td>2.39</td>
<td>17.13 (17.13)</td>
<td>0.75</td>
</tr>
<tr>
<td>Scaffold</td>
<td>84.73 ± 0.17</td>
<td>4.78</td>
<td>17.13 (17.13)</td>
<td>0.74</td>
</tr>
<tr>
<td>FedNova [86]</td>
<td>82.71 ± 0.27</td>
<td>2.39</td>
<td>17.13 (17.13)</td>
<td>0.73</td>
</tr>
</tbody>
</table>

AdaSplit (κ=0.6, η=0.6) | 88.88 ± 0.27 | 9.71           | 5.38 (8.82)      | 0.85     |
AdaSplit (κ=0.75, η=0.6) | 87.11 ± 0.59 | 2.43           | 5.38 (10.88)     | 0.83     |

Table 2: Results on Mixed-CIFAR dataset. AdaSplit achieves improved performance while reducing resource (bandwidth, compute) consumption. This is corroborated by the C3-Score (higher is better).

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>Bandwidth (GB)</th>
<th>Compute (TFLOPS)</th>
<th>C3-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL-basic</td>
<td>67.90 ± 3.52</td>
<td>34.88</td>
<td>1.66 (13.76)</td>
<td>0.59</td>
</tr>
<tr>
<td>SplitFed</td>
<td>71.46 ± 2.13</td>
<td>35.94</td>
<td>1.66 (13.76)</td>
<td>0.62</td>
</tr>
<tr>
<td>FedAvg [56]</td>
<td>91.31 ± 0.49</td>
<td>2.39</td>
<td>11.77 (11.77)</td>
<td>0.79</td>
</tr>
<tr>
<td>FedProx [49]</td>
<td>92.54 ± 0.48</td>
<td>2.39</td>
<td>11.77 (11.77)</td>
<td>0.81</td>
</tr>
<tr>
<td>Scaffold</td>
<td>87.30 ± 1.36</td>
<td>4.79</td>
<td>11.77 (11.77)</td>
<td>0.76</td>
</tr>
<tr>
<td>FedNova [86]</td>
<td>88.94 ± 0.32</td>
<td>2.39</td>
<td>11.77 (11.77)</td>
<td>0.77</td>
</tr>
</tbody>
</table>

AdaSplit (κ=0.6, η=0.6) | 91.92 ± 1.88 | 2.85           | 2.38 (4.81)      | 0.89     |
AdaSplit (κ=0.3, η=0.6) | 92.91 ± 0.91 | 6.57           | 2.38 (6.63)      | 0.88     |

5 RESULTS

We report performance on Mixed-CIFAR in Table 2 and Mixed-NonIID in Table 1. For C3-Score, we set the bandwidth and communication budgets to be $B_{max} = 35.94$ GB and $C_{max} = 11.77$ TLLOPS on Mixed-CIFAR and $B_{max} = 84.64$ GB and $C_{max} = 17.13$ TLLOPS on Mixed-NonIID. These values correspond to the highest bandwidth and computation cost among all the methods for the corresponding datasets. The results on both datasets consistently support the following key observations:

1. AdaSplit outperforms other split learning techniques and achieves significantly better accuracy while also reducing bandwidth consumption. For instance, on Mixed-CIFAR (Table 2), in comparison to SL-basic, AdaSplit improves performance by 24% and consumes 89% lower bandwidth. This is corroborated by an increase in C3-Score from 0.59 for SL-basic [27] to 0.89 for AdaSplit.

Furthermore, similar trend is observed on Mixed-NonIID (Table 1) as well. Specifically, AdaSplit achieves accuracy of 88.88 against 84.67 for SplitFed while consuming 75 GB less bandwidth. The corresponding trend is also captured by C3-Score which is 0.85 for AdaSplit against 0.73 for SplitFed [79].

2. AdaSplit makes split learning a competitive alternative to federated learning. On both datasets, we observe that AdaSplit consistently achieves higher (or similar) accuracy with significantly lower client compute and similar bandwidth. For instance, on Mixed-NonIID, AdaSplit achieves 87.11% accuracy with 2.43 GB bandwidth and 5.38 TFLOPS compute. In comparison, the closest federated learning baseline, FedProx, achieves 85% accuracy but consumes 17.13 TFLOPS (3x more than AdaSplit) and with similar bandwidth (2.39 GB). This is corroborated with an improved C3-Score of 0.85 for AdaSplit and 0.75 for FedProx.

3. AdaSplit consistently provides the best trade-off among all of federated and split learning baselines. For instance, on Mixed-CIFAR, AdaSplit achieves a C3-Score of 0.89 with the closest federated learning baseline (FedProx) [49] at 0.81, FedAvg [56] at 0.79 and SplitFed at 0.62. Furthermore, similar trend is observed on Mixed-NonIID as well. Specifically, AdaSplit achieves a C3-Score of 0.85 with the closest baseline FedProx at 0.75, Scaffold [35] at 0.74 and SL-basic [27] at 0.72.

4. AdaSplit can adapt to variable resource budgets. From results on Mixed-NonIID (Table 5), we can see that given a higher communication budget (13.36 GB), AdaSplit can further improve accuracy to 89.77% which corresponds to a 5% improvement over FedProx [49]. Figure 1 shows trade-offs that AdaSplit can achieve for accuracy by varying bandwidth (and compute) budget. We note that the respective trade-off curves for bandwidth (and compute)
6 DISCUSSION

In this section, we conduct a sensitivity analysis of key design choices in AdaSplit and analyze the consequent impact on accuracy and resource consumption. Results validate the ability of AdaSplit to efficiently adapt to variable resource budgets. Unless specified otherwise, the hyperparameters used are $\kappa = 0.6$, $\eta = 0.6$, $\mu = 0.2$.

1 Varying Size of Client Model: Table 3 presents results from varying the number of layers on client for Mixed-CIFAR10 dataset. We observe that Computation on client increases monotonically with the number of client layers. We also observe a decrease in Communication cost as evident from lower bandwidth. This can be attributed to the convolution design of the model where split activations becomes smaller with depth (reducing payload $P_{ij}$). Also, we note marginal gain in performance for larger server model since it provides more parameters for Collaboration. We observe similar trends on Mixed-NonIID and include results in the appendix. Hence, AdaSplit adapts to variable client computation budgets.

2 Varying Duration of Local Phase: Table 4 presents results from varying $\kappa$ on Mixed-CIFAR10 dataset. We observe that Communication cost decreases as $\kappa$ increases. This is because $P_{ij} = 0$ for all rounds $r < \kappa$ on given client $i$. Computation cost of the server also decreases on increasing $\kappa$ though client compute is unchanged. Note marginal decrease in accuracy since higher $\kappa$ allows for fewer collaboration iterations on server. Specifically, increasing $\kappa$ from 0.3 to 0.9 decrease accuracy from 89.80% to 87.11%, while bandwidth falls drastically from 17.22 GB to 2.43 GB. This is corroborated on Mixed-NonIID as shown in Table 5. Hence, AdaSplit adapts to variable communication and server computation budgets.

3 Eliminating Gradient Dependence: Table 5 studies the impact of training client model without gradient from server on Mixed-CIFAR10 dataset. We observe Communication cost decreases significantly with bandwidth reduced by one-half. We observe accuracy is generally insensitive though there is slight increase in

<table>
<thead>
<tr>
<th>$\kappa$</th>
<th>Accuracy</th>
<th>Bandwidth (GB)</th>
<th>Compute (TFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>92.91 ± 0.91</td>
<td>6.57</td>
<td>2.38 (6.63)</td>
</tr>
<tr>
<td>0.45</td>
<td>90.97 ± 1.02</td>
<td>4.72</td>
<td>2.38 (5.72)</td>
</tr>
<tr>
<td>0.6</td>
<td>89.77 ± 1.62</td>
<td>3.56</td>
<td>2.38 (4.81)</td>
</tr>
<tr>
<td>0.75</td>
<td>88.62 ± 3.68</td>
<td>2.15</td>
<td>2.38 (3.90)</td>
</tr>
<tr>
<td>0.90</td>
<td>88.02 ± 0.91</td>
<td>0.89</td>
<td>2.38 (2.98)</td>
</tr>
</tbody>
</table>

Table 4: Results on Mixed-CIFAR10. Varying duration of local phase ($\mu$) enables AdaSplit to adapt to variable communication and server computation budget.

In each Accuracy cell, Row-1 trains client with $L_{client}$ and Row-2 trains client with $L_{client} + L_{server}$. Accuracy is largely insensitive to server gradient across various $\kappa$ standard deviation. Hence, AdaSplit adapts to variable communication budget and provides consistent performance.

4 Further Reducing Payload Size: While we sparsify server model parameters to improve collaboration in AdaSplit, here we also consider sparsification of split activations to reduce communication payload. Specifically, we train the client model with an additional $L^1$ regularizer that regulates magnitude of split activations. Results are presented on Mixed-NonIID in Table 5. Computation remains unchanged. Communication decreases as payload ($P_{ij}$) becomes sparse. This results in fall in accuracy which highlights worsening collaboration. For instance, AdaSplit can train with only 0.76 GB of bandwidth and achieve 85.79% accuracy. From Table 1, FedProx achieves 85.09% and consumes 2.39 GB budget. Hence, AdaSplit adapts to extremely low communication budgets.

7 RELATED WORK

In this section, we review the literature in distributed and collaborative deep learning broadly, and specific research in split learning.

Parallelizable Deep Learning. For centralized machine learning, some parallelization methods have been proposed to enable training on large-scale data sources. Data Parallelism [33] based distributed ML simulates large mini-batch training by splitting data among multiple identical models and training each model on a shard of the data independently. The key challenge is to ensure consistency of the global model by synchronizing across multiple model copies. This is achieved via i) Synchronous Optimization -
which synchronizes after every minibatch [11, 15], resulting in high communication overhead, and ii) Asynchronous Optimization which does global synchronization less frequently [16], improving communication but at a cost of poor sample efficiency due to staleness of model copies. Model Parallelism splits the model parameters over multiple processors either to improve parallelization [6] or when it is too large to fit in the memory of a single device [30, 44, 73]. This is increasingly common as the size of the state-of-the-art models is exploding (to billions of parameters [8, 67]), which can be challenging for data parallelism. However, model parallelism has its challenges including i) high communication costs, due to activations communicated between devices for each mini-batch and ii) device under-utilization, due to inter-device control-flow dependency during forward and backward propagation [5]. Local Parallelism alleviates the staleness issue by eliminating the control flow dependency between devices. The key idea is to perform training using intermediate local objectives. Various local learning configurations are proposed where a local objective can be used greedily for each layer [4, 51], for a set of layers on each device [41] or layers across multiple devices [25]. From a comparison standpoint, federated learning setup is close to data parallelism, split learning extends model parallelism to data-parallel setups and the proposed AdaSplit extends local parallelism to data-parallel setups.

**Distributed Deep Learning.** This broadly involves training deep networks with data that is distributed across multiple devices (or clients) and a coordinating server. The two main paradigms for collaborative learning are Federated Learning (FL) [34, 55] and Split Learning (SL) [27, 65]. This work is focused on the split learning paradigm, with particular interest in mechanisms to improve the efficiency of communication and collaboration. However, here, we contextualize relevant literature in both federated and split learning along the key design choices introduced in Sec 2. D1: Communication FL generalizes data parallelism (to each client) and the entire model trained locally on-device at the data-owning client. While the computation at the server involves averaging of model parameters, computation at the client-side requires training ML model at every round. This can be a bottleneck for clients that are limited by computation. Some recent works have attempted to address this problem by training on a subset of parameters [18] or performing pruning [46, 91] on the trained model. All such methods are limited by the extent of pruning that can be performed and require the full model to be trained on the device. In contrast, SL distributes the model parameters between clients and servers, this allows reduced as well as tunable computation load distribution between client and server. Several architectures for parameter distribution are discussed in [82]. Our proposed AdaSplit allows to further improves computation by eliminating gradient flow and training server intermittently. D2: Communication In FL, client and server communicate once every training round and this is executed through model weights or gradients of the locally trained model copies. This communication cost scales with the size of the model and the number of clients in the system, and is a major challenge for deploying in resource-constrained setups. Various methods have been proposed to reduce this cost through model compression on client [28, 39, 54], client subset selection [3, 13, 60] as well as greedy federated training of client models [58, 60]. In SL, the client and server communicate in every training iteration using minibatch activations. While each payload is small (and relatively independent of model size), a high frequency of communication is a challenge. In this work, locally-parallel AdaSplit reduces communication costs by alleviating the need for high-frequency server-to-client gradient flow. D3: Collaboration For Federated Learning, while FedAvg [56] performs poorly with non-iid clients, various techniques have been proposed for working in such heterogeneous setups [47, 90, 92]. Similarly, SL-basic and SplitFed perform poorly in non-iid setups, resulting in worsened communication cost and accuracy. To the best of our knowledge, no method has been proposed to tackle this issue. AdaSplit alleviates the same through parameter sparsification on the server.

**Split Learning: Research and Applications.** While most of the recent works [34] in federated learning try to address the aforementioned D1, D2, and D3 aspects, the clients are still restricted by the requirement of executing the complete computation graph on the device. This restriction is circumvented by split learning [27] that revamps the federated learning architecture by operationalizing model parallelism for multi-institution collaborative learning. In split learning, clients share activations produced by a neural network instead of the weight or gradients as done in federated learning. Different architectures for forward and backward propagation are illustrated in [83]. This architecture adds flexibility to the client-side computation by distributing it with the centralized server. A rigorous comparison between FL and SL have been made across several dimensions by [80] and other aspects have been explored such as communication [75], healthcare applications [22], privacy [36], IOT [21] and frameworks [85]. Several recent works have integrated federated and split learning architecture such as SplitFed [79], SplitFedV3 [22] and FedSL [1]. In addition to the computational and communication benefit, split learning allows distributed and privacy-preserving prediction that is not possible under the federated learning framework. Consequently, several works have used split learning for inference to build defense [7, 45, 50, 57, 61, 69, 70, 74, 84, 87] and attack mechanisms [32, 53, 64]. Some of these benefits have led to applied evaluation of split learning for mobile phones [62], IoT [37, 38, 63], model selection [71, 72] and healthcare [65].

**8 CONCLUSION**

We introduce AdaSplit, a technique for scaling distributed deep learning to low resource scenarios. AdaSplit builds upon the split learning framework and reduces a) computation by eliminating client dependence on server gradient and training the server intermittently, b) bandwidth by reducing payload size and communication frequency between client-and-server, and c) improves performance by constraining each client to update sparse partitions of server model. To capture and benchmark this multi dimensional nature of distributed deep learning, we also introduce C3-Score, a metric to evaluate performance under resource budgets. We validate the effectiveness of AdaSplit under limited resources through extensive experimental comparison with strong federated and split learning baselines. We also present sensitivity analysis of key design choices in AdaSplit which validates the ability of AdaSplit to provide adaptive trade-offs across variable resource budgets.